Efficient Implementation of Rendezvous

A. SCHIPER, R. SIMON, PH. DESARZENS, AND J.-A. SENGSTAG
Ecole Polytechnique Fédérale, Laboratoire de Systèmes d'Exploitation, MA-Ecublens, CH-1015 Lausanne, Switzerland

In this paper we present a simple and efficient implementation of the rendezvous developed for a small multiprocessor system (6–10 processors). At first, we present the communication primitives with their traditional implementation. Secondly, we discuss in details our implementation and the communication architecture needed. We conclude with an evaluation showing that our implementation needs a maximum of four context switches to realize a rendezvous, which can be considered as an ideal result.

Received July 1987, revised March 1988

1. INTRODUCTION

This paper presents a simple and efficient implementation of the rendezvous communication protocol developed for a multiprocessor system made of 6–10 sites. A site consists of a processor with its own local memory. The sites are connected by a bus. It is reasonable to think that these systems will become available as workstation in few years. This is due to the increasing capacity of chips and their decreasing price. On such system the OS would run on one processor, and applications could ask the OS for free processors. This is similar to the idea developed in the Amoeba distributed system,[7] which consists of workstations connected to a pool of processors. Each application may ask for processors of the pool.

To take advantage of all the CPU power of multiprocessor systems it is important to use an efficient communication protocol. The paper will show both the software and the architecture used to implement efficient rendezvous-based communications. The implementation will be described using the language Modula-2.[8]

2. COMMUNICATION PRIMITIVES

There are two classes of communication primitives, synchronous and asynchronous. Execution of a synchronous primitive blocks the processes until the realisation of the communication. In asynchronous communications, the sender keeps executing without waiting for the receiver. In the latter case the communication kernel has to buffer the messages. We have chosen to implement synchronous primitives for simplicity and efficiency. Doing this, we join the actual trend outlined by CSP,[5] OCCAM,[3] ADA[2] and the V-Kernel.[1] And we make our own the slogan 'Messages are for communication and processes are for concurrency'.[1]

The synchronous communication primitives of our protocol are:

- Send (in destination, in message);
- SendAndWaitReply (in destination, in message, out answer);
- Receive (out sender, out message);
- Reply (in sender, in answer).

The 'Send–Receive' communication realises the usual rendezvous ('sender' is an output parameter of the Receive primitive taking the value of the sending process). The 'SendAndWaitReply–Receive–Reply' communication (similar to the V-Kernel primitives) realises a procedure-call type of synchronisation:

- process P executes SendAndWaitReply(Q, m) and is stopped;
- process Q executes Receive (sender, m) and gets the message of P in the variable m and 'sender' gets the process Id of P.
- process Q executes Reply (sender, answer), which releases P and gives him the answer of Q. Both P and Q continue their execution independently from now on.

Note that the primitive Send is not necessary, as it is possible to send an empty reply message just after Receive, to achieve the Send functionality using the SendAndWaitReply primitive. We will keep both primitives for the sake of clarity. It is also worthwhile to observe that the two sends (Send, SendAndWaitReply) correspond to the two models of process relationship:

- producer–consumer relationship of a pipeline parallelism, using a communication of type 'Send–Receive';
- client–server relationship, using a communication of type 'SendAndWaitReply–Receive–Reply'.

3. TRADITIONAL IMPLEMENTATION

3.1 Ideas of the traditional implementation

A remote rendezvous implementation must take into account the characteristics of the transmission medium. In our case we can make the following assumptions concerning the transmission of messages on the bus connecting the different sites: (1) messages are neither lost nor corrupted; (2) messages are delivered in emission order. Using these assumptions a traditional implementation of the 'Send–Receive' primitives would be as follows. The initial message m is sent to the destination site and stored in a queue of messages for the destination process. If this process is ready to receive a message, the message is immediately delivered and a 'proceed' message is sent to the sender process, allowing him to proceed. If the destination process is not ready to receive the message m, the proceed message will be sent later. In case there is no free buffer on the destination site when the message arrives, the message is simply discarded. This introduces the need for a timeout in the sender process. If the timeout expires without receiving 'proceed', the message m must be retransmitted. As a consequence,

messages can be received twice (or more). This introduces the need to number the messages to distinguish between new and retransmitted messages.

The implementation of the 'SendAndWaitReply–Receive–Reply' primitives follows the same scheme with one exception; the 'proceed' message is replaced by the reply message.

This protocol costs two packets in normal circumstances (initial message + proceed message or initial message + reply). In case of a timout caused either by a discarded message (no free buffer) or by a heavily loaded destination CPU (receiving process obtaining few CPU time), an unbounded number of packets can be sent. To avoid this situation, a three-packet protocol can be used (adapted from Ref. 4). The destination site immediately sends a 'received–ack' packet after receiving a message, informing the sender that the message has been queued on the destination site. Note that this protocol does not suppress the need for timeout on the sender side; it allows the reduction of the timeout period (which covers now only the time needed to transmit the message) and reduces the number of times that timeout will occur. So the three-packet protocol is simply (1) initial message from sender site to destination site, (2) receive–ack from destination to sender, and (3) proceed or reply from destination to sender.

3.2 Drawbacks of the traditional implementation

The two-packet protocol suffers from poor performances in the case of a heavily loaded destination CPU. A high timeout value on the sender site reduces the number of message retransmissions but increases the communication time in the case of a discarded message. A low timeout value increases the number of message retransmissions. The three-packet protocol does not suffer as much from performance degradation.

Another drawback of the traditional implementation is the need for a server process, responsible for receiving the incoming messages and for transferring them to the destination process's queue. This requires two unnecessary context switches, the first to receive the message, the second to return to the interrupted process. A context switch should occur only if the destination process is ready to treat the message!

The implementation we propose costs two packets to implement the 'Send–Receive' communication and three packets to implement the 'SendAndWaitReply–Receive–Reply' communication, without suffering from performance degradation in the case of heavy loaded CPUs because the protocol needs no timeout. Concerning the context switches, our implementation has the convenient property of delaying context switch until the incoming message can be treated by the destination process. The idea is to buffer all messages except one on the sending site rather than on the receiving site, as is done traditionally.

4. THE IDEAS OF OUR IMPLEMENTATION

4.1 How to avoid timeouts

Taking into account the assumption of a reliable transmission, timeouts are only needed because the sending process cannot be sure that a buffer is available on the destination site. Timeouts and all their drawbacks can be suppressed if a message is sent only when a buffer is available on the destination site. To reach this objective we allocate statically 'reception buffers' to each source site A on each destination site B. When a message is sent from site A to site B, site A knows the status of the buffer on B to which the message is sent. If the buffer is empty, the message can be sent; if the buffer is full the emission of the message is delayed. The source site will be informed that the buffer has been emptied by a special message called 'release'. At that moment the delayed message is sent.

One reception buffer allocated to each site A on each site B is not enough. This could lead to implementation deadlocks, as shown by the example in Fig. 1. Suppose the buffer of A on site B contains message m1 sent by process Pa1 to Pb1. The buffer will not be emptied until Pb1 executes 'Receive(...)'. But before executing 'Receive(...)', process Pb1 has to communicate with process Pa2. Unfortunately this rendezvous cannot take place until Pa2 had done his rendezvous with Pb2. But Pa2's message m2 cannot be sent until the reception buffer of A on site B, which contains m1, is emptied.

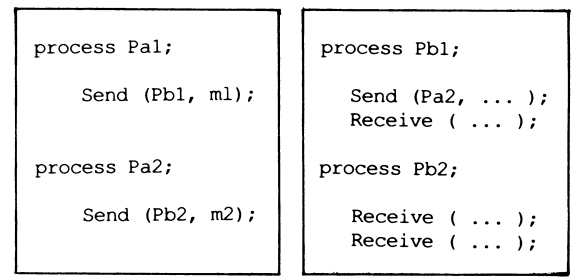


Figure 1. Illustration of implementation deadlock when site A owns only one reception buffer on site B.

The deadlock is due to the fact that the logically possible rendezvous between processes Pa2 and Pb2 cannot take place due to implementation constraints. To avoid these deadlocks it is necessary and sufficient that each site A own on each site B an amount of p reception buffers, where p is the number of processes on site B. Let us name these buffers

reception_ofA_onB [1], ..., reception_ofA_onB [p]

If process Pb on site B executes 'Receive(...)' the following two situations can occur: (1) the buffer reception_ofA_onB [Pb] is full, so a rendezvous can take place between Pb and a process on site A; (2) the buffer reception_ofA_onB [Pb] is empty, meaning that no rendezvous with a process on site A can take place for the moment.

If process Pa on site A executes 'Send (Pb, mess)', the following two situations are possible: (1) the buffer reception_ofA_onB [Pb] (which resides on site B) is empty and the message is sent; (2) the buffer reception_of A_onB [Pb] is full and the message cannot be sent for the moment. This causes what we call a 'delayed Send'. We will see in Section 5 how the delayed Send is treated. Observe that not sending the message does not slow down the system because its receiver is busy anyway.

Taking all together, p*s reception buffers are needed

on each site, where s is the number of sites of the multiprocessor and p the maximum number of processes on each site. (In order to simplify explanations, we do not distinguish between inter- and intra-processor communications.) These p*s buffers could seem very important. The following should be noted:

- in our implementation 256K of memory are dedicated to the communication buffers (reception and other needed buffers, see Appendix A). Considering 1K bytes for each reception buffer and 8 sites, this implementation allows 15 processes per site;
- for our first application, which consists of a parallel Modula-2 compiler, only a few processes per site are needed (less than 5);
- the important point, beyond these numbers, is that you can mix both the traditional implementation and our implementation using statically allocated buffers. A limited number of intensively communicating processes could use the efficient implementation, the others (unlimited number) the traditional implementation.

4.2 How to avoid unnecessary context switches

The way to avoid unnecessary switches is very simple. A dedicated interrupt vector is allocated to every process, suppressing the need for the server process of the traditional implementation. Each time a process has to wait (after a Send waiting for the release message, after a Receive waiting for an incoming message or after a SendAndWaitReply waiting for the Reply), it waits on its interrupt, appropriately setting an interrupt mask composed of 3 bits called 'receive', 'release' and 'reply'. For example, to wait exclusively on an incoming message simply consists of setting the interrupt mask to (receive: = 1, release: = 0, reply: = 0). The incoming message will contain a similar mask, identifying the message type. When a message arrives, its mask is compared to the interrupt mask (logical and). If the result is different from zero, an interrupt is generated. More information about this point can be found in Appendix A (communication architecture) and in Appendix B (implementation of the communication primitives).

4.3 The protocol

The 'Send–Receive' communication results in the following protocol: (1) initial message sent from sender site to destination site; (2) release message sent from destination site to sender site to inform that the destination buffer has been released and that the rendezvous is done (the release message plays here the role of the 'proceed' message).

The 'SendAndWaitReply–Receive–Reply' results in the following protocol: (1) initial message sent from sender site to destination site; (2) release message sent from destination site to sender site to inform that the destination buffer has been released; (3) reply message sent from destination site to sender site. Note that the release and the reply messages must both exist, because the reply messages are not necessarily sent in the order corresponding to the reception of the messages (a server could receive message m1 then message m2, but reply first to m2).

We now have all the elements to describe precisely the communication primitives. Their implementation is in Appendix B. The implementation of Receive and Reply is straightforward. On the other hand Send and SendAndWaitReply need explanations. This will be done in the next section.

5. DELAYED SEND

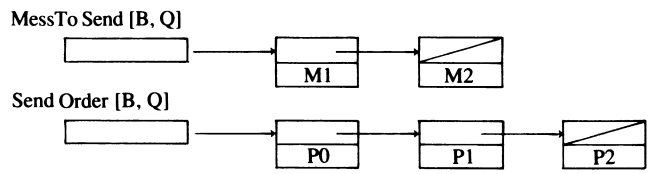
As we said in Section 4, messages cannot be sent as long as the reception buffer is full. This causes what we call a 'delayed Send'. The aim of implementing delayed Send is reduction of the number of context switches. Consider the following situation. A process Pa on site A cannot send a message m to a process Pb on site B because the reception buffer 'reception_ofA_onB [Pb]' is full (meaning also that the destination process is not ready to receive the message of process Pa). Who will later send the message m? If it is Pa, this requires two context switches (the first to activate Pa to send the message, the second when Pa waits for the release message). A better implementation can sometimes avoid these two context switches. The idea comes from the following observation: if the reception buffer is full, this means that another process Px on site A has sent a message to the same destination process Pb.

Suppose first that Px has executed the primitive 'Send'. When Px receives the release packet corresponding to the end of the rendezvous, Px sends Pa's message, before continuing his execution, thus avoiding unnecessary context switches!

Suppose now that Px has executed the primitive 'SendAndWaitReply'. The situation is here a little more complicated; we cannot avoid the two context switches. However, the two context switches needed to send Pa's message are used to activate process Px and not Pa, because the destination process Pb cannot know the existence of the delayed Send for process Pa. Considering the protocol of paragraph 4.3, process Px receives two messages, the release and the reply message. In the absence of any delayed Send the release message is unimportant and should cause no interrupt. So initially Px sets its interrupt mask to (receive: = 0, release: = 0, reply: = 1) and will only be interrupted by the arrival of the reply message. But the occurrence of a delayed Send for process Pa renders the release message important. Therefore Pa sets the release bit of Px's interrupt mask, causing Px to be interrupted by the release message, which then allows Pa's message to be sent by Px.

Two data structures (shown in Fig. 2) are needed on each site to implement the delayed Send:

- MessToSend [site#, prss#], where each element of the array is a list of messages to send to process 'prss#' on site 'site#';



State of the lists for a scenario : P0 occupies the reception buffer while P1 and P2 have not been authorised to send their respective messages M1 and M2 (all processes send to the same receiver). P0 will send message M1 and P1 message M2.

Figure 2. Data structures for delayed Send.

Communication	Delayed Send	Receiver	Total context switch
Send–Receive		Not waiting	2
		Waiting	4
SendAndWaitReply–Receive–Reply	Yes	Not waiting	4
	Yes	Waiting	(6) Virtually impossible
	No	Not waiting	2
	No	Waiting	4

Figure 3. Number of context switches.

- SendOrder [site#, prss#], where each element of the array is a list of processes corresponding to the order of execution of the Send or SendAndWaitReply primitives to process 'prss#' of site 'site#'.

6. EVALUATION

Our objective was to build an efficient implementation of rendezvous. The protocol proposed aims at efficiency by using statically allocated buffers in order to reduce the number of context switches and to suppress the need for timeouts. The suppression of timeouts ensures good response times for rendezvous even when the multiprocessor is heavily loaded. The efficiency of rendezvous is estimated by counting the number of context switches needed for a complete communication between two processes. The results of our implementation should be compared to the 4 context switches, which can be considered as an ideal result:

- 2 context switches on the sender site (1 to block the process waiting for the receive or reply message, 1 to activate the process when the receive or reply message arrives);
- 2 context switches on the receiver site (1 to block the process waiting for a message, 1 to activate the process when the message arrives).

Considering first our implementation of a 'Send–Receive' communication. This communication costs at most 4 context switches (Fig. 3):

- 2 context switches on the sender site (the 2 context switches mentioned above);
- 0 or 2 context switches on the receiving site: 0 if the message arrives before Receive is executed (corresponds to 'Receiver not waiting' in Fig. 3), 2 otherwise ('Receiver waiting').

Consider now a communication of type 'SendAndWaitReply–Receive–Reply'. We get the following results (Fig. 3):

- 2 or 4 context switches on the sender site (4 in case of delayed Send, including the 2 context switches explained in Section 5 needed to send the delayed message);
- 0 or 2 context switches on the receiving site, 0 if the message arrives before Receive is executed ('Receiver not waiting'), 2 otherwise ('Receiver waiting').

However, it is important to notice that delayed Send virtually implies arrival of messages before execution of the corresponding Receive. Consider the following situation.

Site A:

- process Pa1 executes 'Send (Pb, m1)'
- process Pa2 executes 'Send (Pb, m2)', which results in a delayed Send (reception buffer contains Pa1's message)

Site B:

- process Pb executes following sequence Receive(..., m1); treatment of m1; Receive(..., m2);

The delayed message m2 is shipped to the destination process Pb1 as soon as Pb1 has received the message m1 and has sent the corresponding release message. The time needed to transmit the release message and the next message m2 is normally much less than m1's treatment time by process Pb (starting after the emission of the release message). Therefore a delayed Send virtually implies the arrival of the message before execution of the corresponding Receive. So the result is a total of 4 context switches.

The global result is a maximum of four context switches for both communication types, which can be considered as ideal. Although we have not yet implemented the protocol (which will start very soon), the communication is expected to be efficient.

Acknowledgement

We wish to thank Jorge Eggli for his helpful comments during the development of the rendezvous protocol.

REFERENCES

1. D. R. Cheriton, The V-kernel: a software base for distributed systems. *IEEE Software*, pp. 19–42 (April 1984).
2. *Ada Programming Language*. United States Department of Defense, Washington (1983).
3. Inmos, *Occam Programming Manual*. Prentice-Hall, Englewood Cliffs, N.J. (1984).
4. N. D. Gammage, R. F. Kamel and L. M. Casey, Remote rendezvous. *Software-Practice and Experience* **17** (10) 741–755 (1987).
5. C. A. R. Hoare, Communicating sequential processes. *Comm. ACM* **21** (8), 666–667 (1978).
6. J.-D. Nicoud, Video RAMs: structure and applications. *IEEE Micro* **8** (1), 8–27 (1988).
7. A. S. Tanenbaum and R. van Renesse, Distributed operating systems. *ACM Computing Surveys* **17** (3), Heidelberg, 419–470 (1985).
8. N. Wirth, *Programming in Modula-2*. Springer-Verlag, (1985).

APPENDIX A. COMMUNICATION ARCHITECTURE

A.1 Generality

Each site of the multiprocessor consists of a MC 68020 processor and 2M bytes of local memory. The communication buffers are allocated in an extra 256K bytes of video-ram, a dual-port memory.[6] One port allows the processor to access the communication buffers; the other port connects the video-ram to the multiprocessor bus. The hardware sends messages from the video-ram of one site directly to the video-ram of another site.

All the buffers needed for the rendezvous implementation are allocated in this video-ram. They are:

- the s*p reception buffers (s = number of sites, p = maximum number of processes per site), to receive the initial message;
- the p reply buffers (one per process), to receive the reply messages;
- the release buffers, to receive the 'release' messages. As one site owns p reception buffers on each site, p*s release buffers are needed.

As already mentioned, in the current implementation s = 8 and thus p = 15.

A.2 Current interrupt realisation

For economic reasons there is for the moment only one interrupt vector, and the interrupt handling mentioned in paragraph 4.3 is implemented through ROM routines. These routines allow the communication kernel written in Modula-2 to see the interrupt handling as described in paragraph 4.3. This works in the following way. When a message is received on a site, the address of the message's buffer is inserted by the hardware in a FIFO. As long as this FIFO is not empty, an interrupt is generated. This interrupt is treated by the ROM routines, which doesn't cause a Modula-2 context switch. Depending on the message's interrupt mask and the vector's interrupt mask, these routines generate the pseudo-interrupt awaited by the Modula-2 processes.

APPENDIX B. IMPLEMENTATION OF THE COMMUNICATION PRIMITIVES

The communication primitives are expressed, taking into account the architecture mentioned in Appendix A. With this communication architecture we can make the assumption that no message arrival is 'seen' while a process is executing within the kernel (interrupt masked→ROM routines cannot be called).

To express the communication primitives, let us introduce the following names for the communication buffers:

- RECEPTION [site#, process#], into which the messages for process# from a sender on site# are received. Each buffer has a 1-bit full/empty status.
- ANSWER [process#], into which the reply messages for process# are received. Each buffer has a 1-bit full/empty status.
- RELEASE [site#, process#], into which the released messages are received. A message in RELEASE [Sj, Pk] on site Si means that the buffer RECEPTION [Si, Pk] on site Sj has been emptied.

To illustrate the usage of these buffers, consider a communication of type 'SendAndWaitReply–Receive–Reply' between a sender process Q on site Sq and a destination process R on site Sr. Let us suppose that Q is the process number Pq of site Sq and that R is the process number Pr of site Sr:

(1) Q executes SendAndWaitReply:
the initial message is sent to the buffer RECEPTION[Sq, Pr] on site Sr;
(2) R frees the reception buffer:
the release message is sent to the buffer RELEASE[Sr, Pr] on site Sq;
(3) R sends the reply:
the reply message is sent to the buffer ANSWER[Pq] on site Sq.

Finally, here are some complementary explanations:

- each message has a header which contains besides other information an interrupt mask;
- the procedure Ship is used to ship the message (message = header + info);
- 'myself' is used to name the process executing a given procedure;
- initially all buffers are empty except the RELEASE buffers which are considered full (meaning that the corresponding RECEIVE buffers are empty);
- the only useful information of the SendOrder queue (Section 5) is the name of its last element. This information will be stored in a variable called LastSendingPrss.

```
PROCEDURE Receive ( VAR sender: PrssName;
VAR messContent: ... );
BEGIN
  IF for all site S, RECEPTION [S,
  myself.prss] is empty
  THEN
    (* no message are already here *)
    myself.mask:=( receive:=1, release:=
    0, reply:=0 );
    Wait on MY interrupt for a Send or
    SendAndWaitReply;
  END (* if *);
  senderSite:=Get the first site number
     such that RECEPTION [senderSite,
     myself] is full;
  sender:=logical name of the sender
     extracted from the message header in
     RECEPTION [senderSite, myself];
  messContent:=content of the message in
  RECEPTION [senderSite, myself];
  myself.mask:=( receive:=0, release:=0,
  reply:=0 );
  (* now send the release message *)
  Initialisation of the header, in
  particular:
    releaseHeader.address:=address of
       RELEASE [myself.site, myself.prss]
       on sender's site;
    releaseHeader.mask:=( receive:=0,
    release:=1, reply:=0 );
  Ship ( releaseHeader, empty content );
END Receive;
```

Figure 4. Receive.

```
PROCEDURE Reply ( client: PrssName;
answerContent: ... );
BEGIN
  Initialisation of the header, in
  particular:
    replyHeader.address:=address of ANSWER
    [client.prss] on client's site;
    replyHeader.mask:=( receive:=0;
    release:=0; reply:=1 );
  Ship ( replyHeader, answerContent );
END Reply;
```

Figure 5. Reply.

```
PROCEDURE Send ( consumer: PrssName;
messContent: ... );
BEGIN
  Initialisation of the header, in
  particular:
    sendHeader.address:=address of
    RECEPTION [myself.site, consumer.prss]
       on consumer's site;
    sendHeader.mask:=( receive:=1;
    release:=0; reply:=0 );
  IF buffer RELEASE [consumer.site,
  consumer.prss] is full
  THEN (* a message can be sent *)
    Set buffer RELEASE [consumer.site,
    consumer.prss] to empty;
    Ship ( sendHeader, messContent );
  ELSE
    (* RECEPTION [myself.site,
    consumer.prss] on consumer's site is
    occupied *)
    Put the header and content of the
       message at the end of MessToSend
       [consumer.site, consumer.prss];
    Set the bit 'release' of the mask
    associated to the interrupt of
    LastSendingPrss;
    (* the release message issued when
       executing the Receive will wake up
       LastSendingsPrss who will send the
       message *)
  END (* if *);
  LastSendingPrss:=myself;
  myself.mask:=( receive:=0, release:=1,
  reply:=0 );
  Wait on MY interrupt;

  (* consumer has received the message, the
     reception buffer is empty. So before
     exiting we check for delayed send *)
  IF MessToSend [consumer.site,
  consumer.prss] not empty
  THEN
    Set buffer RELEASE [consumer.site,
    consumer.prss] to empty;
    Ship ( head element of MessToSend
    [consumer.site, consumer.prss] );
  END (* if *);
  myself.mask:=( receive:=0, release:=0,
  reply:=0 );
END Send;
```

Figure 6. Send.

```
PROCEDURE SendAndWaitReply ( consumer:
PrssName; messContent: ...; answerContent:
   ... );
BEGIN
  Initialisation of the header, in
  particular:
    sendHeader.address:=address of
       RECEPTION [myself.site,
       consumer.prss] on consumer's site;
    sendHeader.mask:=( receive:=1;
    release:=0; reply:=0 );
  IF buffer RELEASE [consumer.site,
  consumer.prss] is full
  THEN (* a message can be sent *)
    Set buffer RELEASE [consumer.site,
    consumer.prss] to empty;
    Ship ( sendHeader, messContent );
  ELSE
    (* RECEPTION [myself.site,
    consumer.prss] on consumer's site is
    occupied *)
    Put the header and content of the
       message at the end of MessToSend
       [consumer.site, consumer.prss];
    set the bit 'release' to the mask
    associated to the interrupt of
    LastSendingPrss;
    (* the release message issued when
       executing the Receive will wake up
       LastSendingPrss who will send the
       message *)
  END (* if *);
  LastSendingPrss:=myself;
  myself.mask:=( receive:=0, release:=0,
  reply:=1 );
  Wait on MY interrupt;

  (* interrupt has occurred, 2
  possibilities: release or reply *)
  IF buffer ANSWER [ myself.prss ] is empty
  THEN
    (* no reply, interrupt due to a release
    message *)
    (* queue MessToSend [consumer.site,
    consumer.prss] cannot be empty *)
    Set buffer RELEASE [consumer.site,
    consumer.prss] to empty;
    Ship ( head of MessToSend
    [consumer.site, consumer.prss] );
    myself.mask:=( receive:=0, release:=
    0, reply:=1 );
    Wait on MY interrupt; (* this time for
    reply *)
  END (* if *);

  (* the reply has been made *)
  myself.mask:=( receive:=0, release:=0,
  reply:=0 );
  Copy the answer from ANSWER [myself.prss]
  to the variable answerContent;
  END SendAndWaitReply;
```

Figure 7. SendAndWaitReply.